



AI 응용시스템의 이해와 구축

8강. 분산학습 + 파이프라인 최적화

—
출석.

Logistics



중간고사: grading in progress (다음주 수업에 지면 배포 예정)

Project: 코멘트 드리고 있습니다.

9강 (5월 7일)부터 대면수업으로 전환

(전체 소프트웨어융합대학원+ 타 대학원)

“혼합”(hybrid)도 가능하나, zoom 셋업이 된 강의실 사용가능함에 한함.

강의실: TBD

Backpropagation Refresh

Building Neural Network for MNIST

Assumption:

- 각 이미지마다 28x28 픽셀로 이루어짐.
- 각 픽셀마다 floating point 로 grayscale을 표현.
- 따라서 각 input data는 784 개의 floating point vector로 표현 가능.

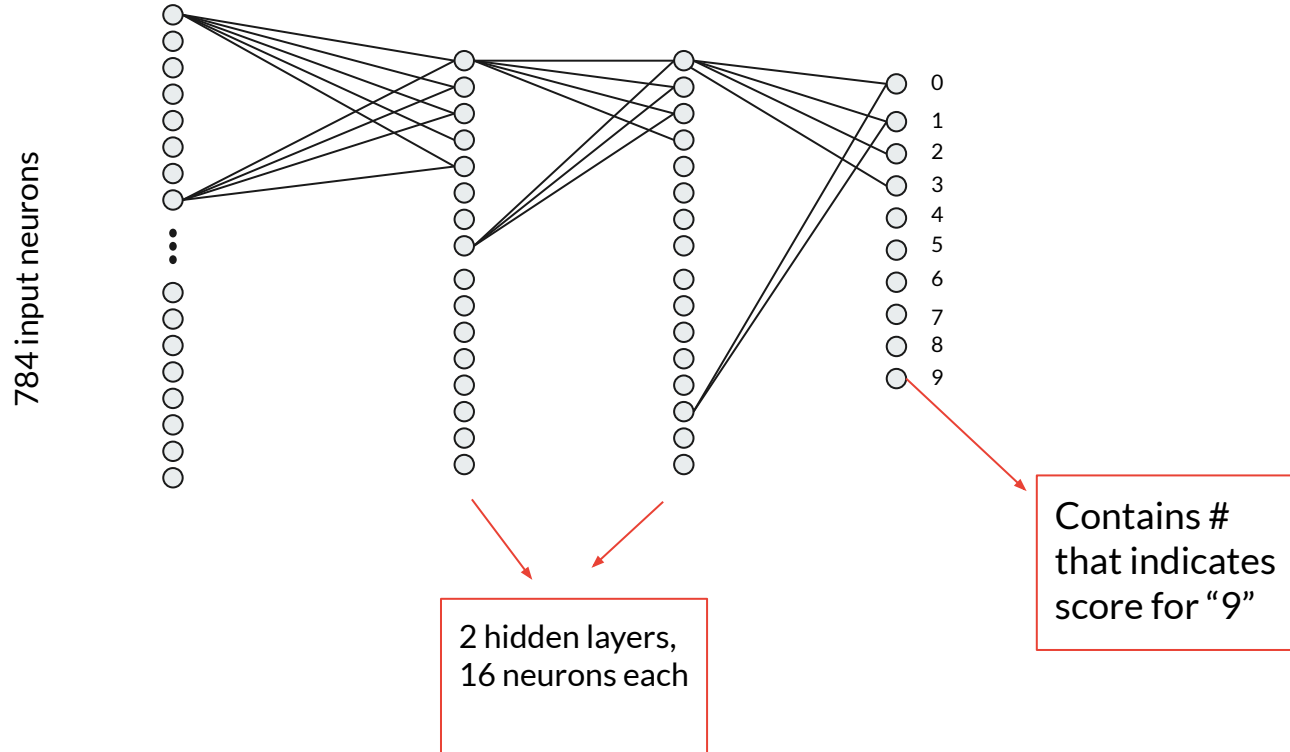


Neuron?

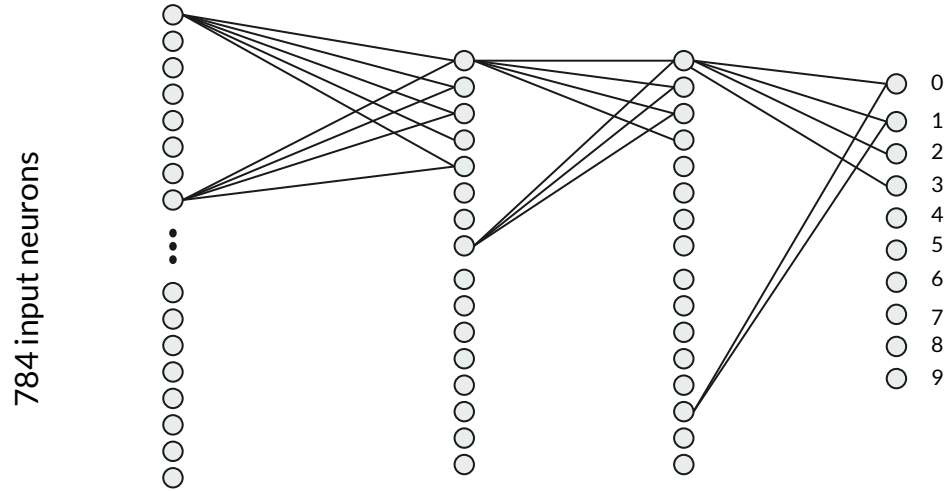


- 각 뉴론마다 하나의 **value**를 저장
- Activation

Neural Network (Multilayer Perceptron)



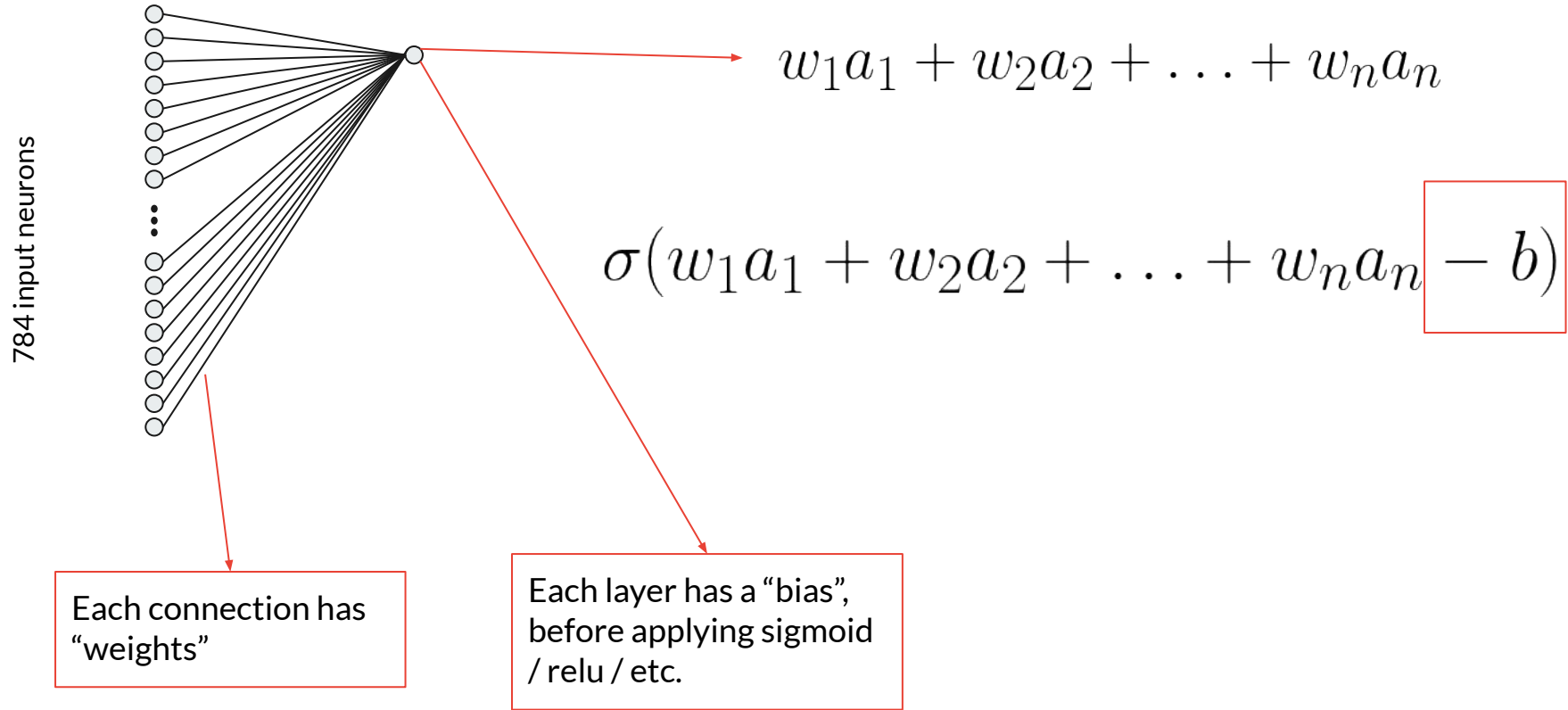
Neural Network (Multilayer Perceptron)



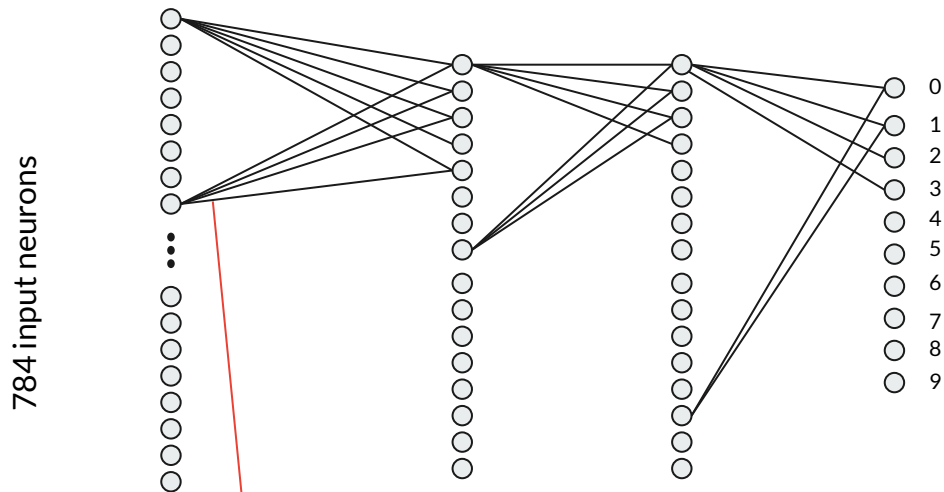
Each layer represents
(hopefully) a subproblem:
E.g. a circle, long edge, ...



Neural Network (Multilayer Perceptron)



Forward Pass



Each connection has
“weights”

Each layer has a
“bias”

$$\sigma(w_1a_1 + w_2a_2 + \dots + w_na_n - b)$$

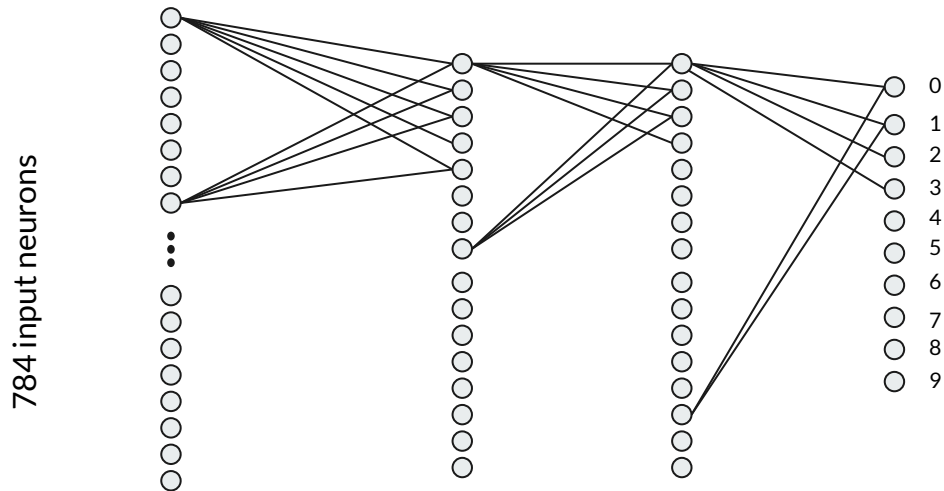
총 파라미터가
몇개?

$784 * 16 + 16 * 16 + 16 * 10$
weights

$16 + 16 + 10$
biases

→ **13,002 parameters**

Forward Pass



Cost function:

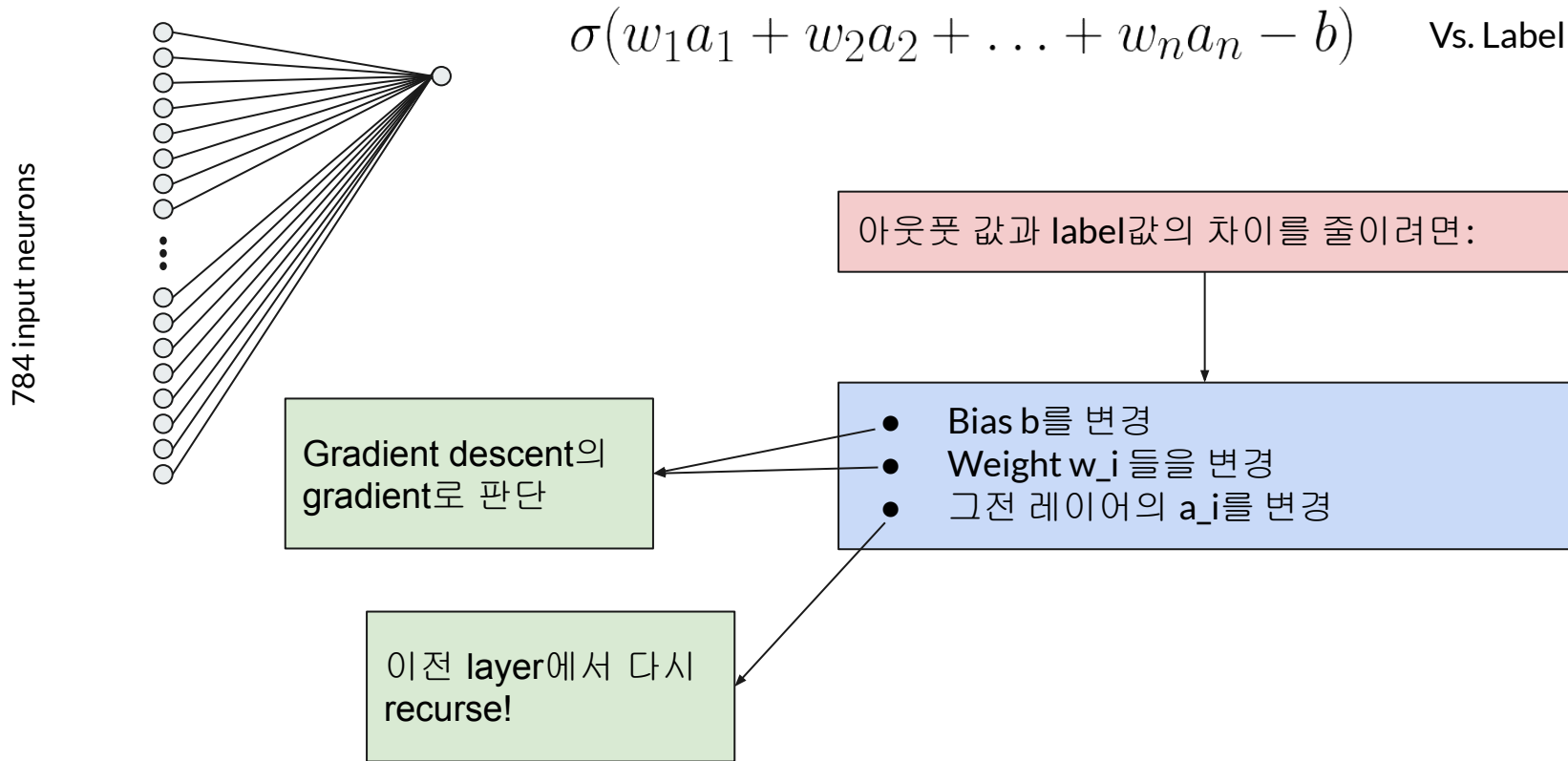
Predict = [0, 0.2, 0.8, ... 0.1]

Label = [0, 0, 1, 0, ..., 0]

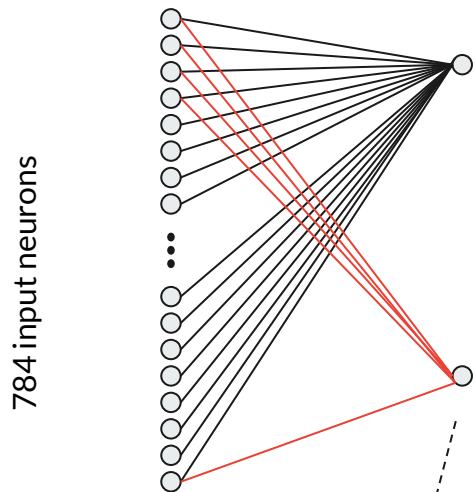
Cost = difference between P, L

모든 데이터셋에 계산하여
평균값 계산 필요!

Backpropagation



Backpropagation



$$\sigma(w_1a_1 + w_2a_2 + \dots + w_na_n - b)$$

아웃풋 값과 label값의 차이를 줄이려면:

- Bias b 를 변경
- Weight w_i 들을 변경
- 그전 레이어의 a_i 를 변경

각 output neuron 하나마다
변경해야 하는 w_i 들과 b 를
계산하여 summarize

각 label data마다 이 필요한
값들을 다 계산해서
summarize
→ "gradient"

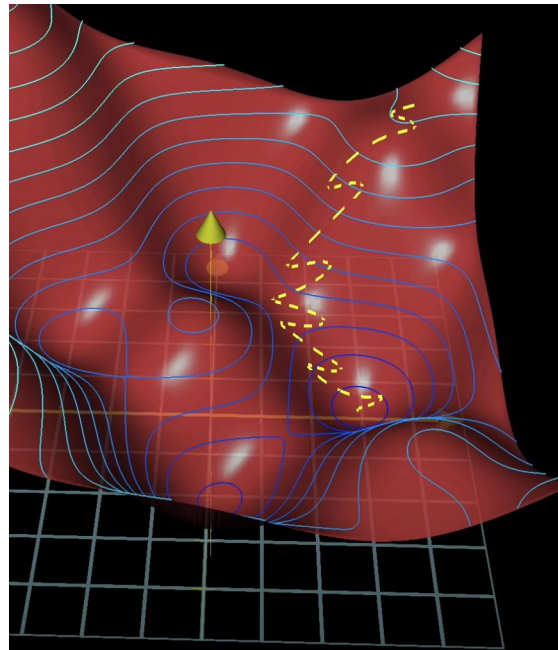
Backpropagation

Gradient를 계산하려면 training data를 전부 사용해야 함.

대신에, train data를 batch로 나누어서 각 batch당 gradient를 계산.

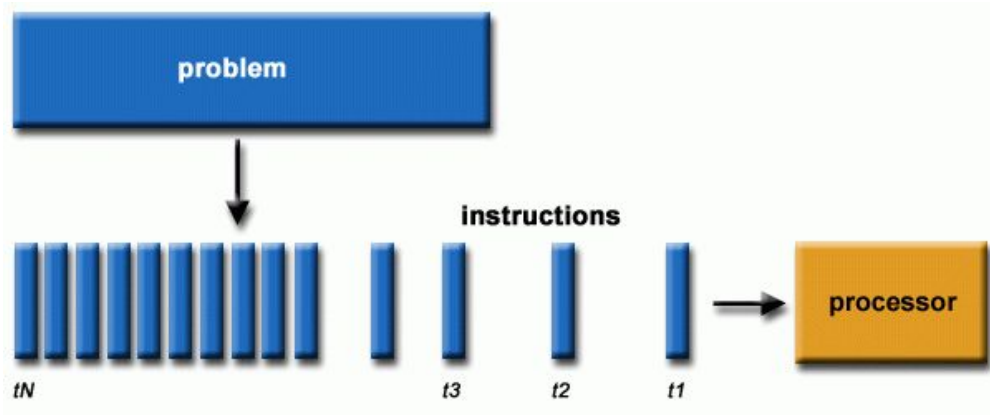
각 batch당 gradient대로 이동 → “stochastic gradient descent”

병렬처리로 계산 가능!



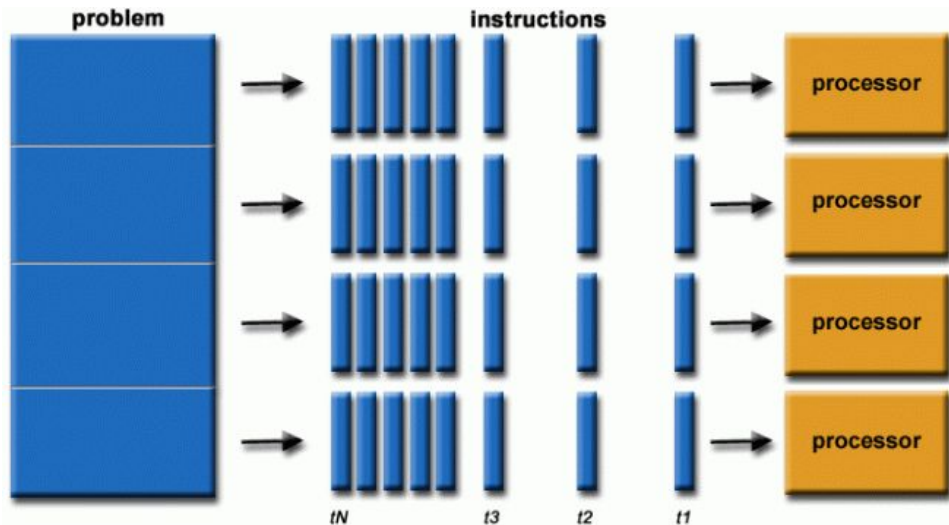
Distributed Machine Learning

Serial Computing



- 1 workstation / server.
- Each workstation → 1 turing machine
- “Algorithm” = sequence of instructions on 1 turing machine.

Distributed (Parallel) Computing



- 문제를 n 개의 subproblem으로 분해
- 가정: 각 subproblem을 독립적으로 계산할 수 있음.

→ search 등에 적합

- Data pipeline (ETL) 등에 사용.
- Spark / Kafka / MapReduce / Hadoop / ...

Parallelism for ML



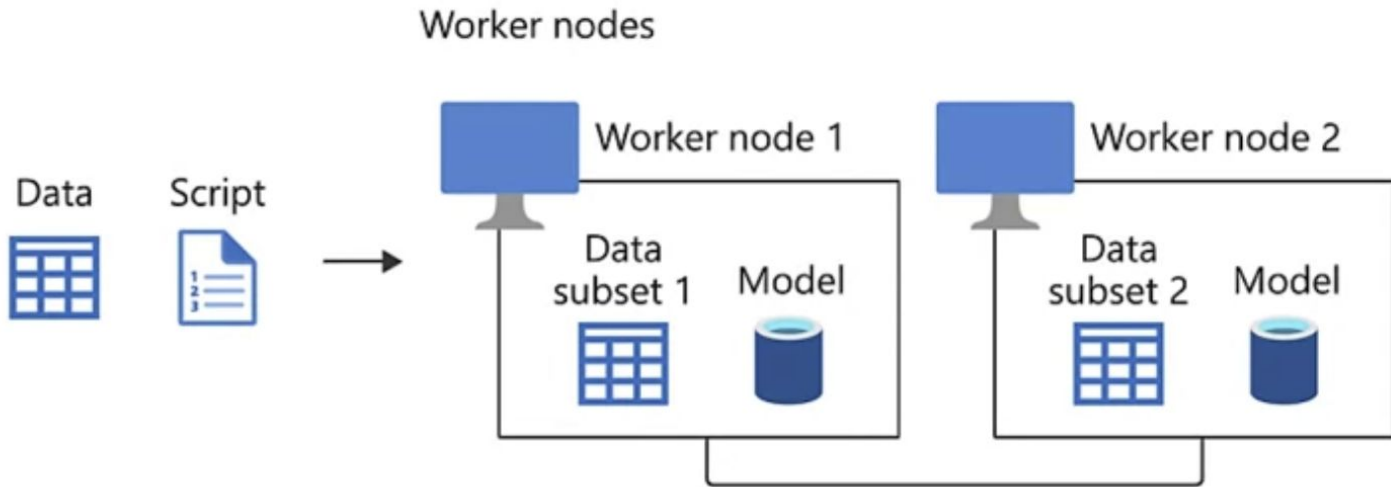
- **Data 병렬처리**

- 모델들이 여러개의 accelerator (GPU/TPU)에 보내져서, 각기 다른 데이터를 처리.
 - 각기 다른 dataset에 **batch training**
 - 커다란 데이터 셋에 대한 **batch inference**

- **Model 병렬처리**

- 모델이 하나의 디바이스 (GPU/TPU)에 로딩하기가 클때, 모델 자체가 여러개의 파티션으로 나뉘 각기 다른 accelerator로 처리.

Data Parallelism



1. 각 노드(worker)가 forward pass에 prediction을 label과 비교
2. label과 비교한 error로 backprop을 사용해서 모델 weight / bias를 업데이트
3. 해당 updated weight / bias를 다른 노드들과 synchronize 필요.

Distributed Training (Data Parallelism)



Synchronous Training

- 모든 worker들이 train & update를 싱크해서 진행
- map/reduce의 “reduce” 아키텍처

Asynchronous Training

- 각자 worker가 train / update를 개별 스케줄로 진행
- Parameter server 아키텍처로 파라미터 공유
- 더 효율적이지만 (compute resource efficient), 정확도나 convergence가 느릴 수 있다.

Distribute-aware model



Framework API

TensorFlow Keras / Estimator 프레임워크가 분산처리 API를 제공.

Custom Training Loop

좀 더 복잡한 아키텍처 (TPU/GPU/edge 믹스)를 위해선 custom training loop을 구현 필요.

Distribute-aware model

Framework API

TensorFlow Keras / Estimator 프레임워크가 분산처리 API를 제공.

```
tf_config = {  
    'cluster': {  
        'worker': ['localhost:12345', 'localhost:23456']  
    },  
    'task': {'type': 'worker', 'index': 0}  
}
```

Distribution Strategy

```
per_worker_batch_size = 64
tf_config = json.loads(os.environ['TF_CONFIG'])
num_workers = len(tf_config['cluster']['worker'])

strategy = tf.distribute.MultiWorkerMirroredStrategy()

global_batch_size = per_worker_batch_size * num_workers
multi_worker_dataset = mnist_setup.mnist_dataset(global_batch_size)

with strategy.scope():
    # Model building/compiling need to be within `strategy.scope()`.
    multi_worker_model = mnist_setup.build_and_compile_cnn_model()

multi_worker_model.fit(multi_worker_dataset, epochs=3, steps_per_epoch=70)
```

- **tf.distribute.Strategy**
 - Tensorflow에서 지원하는 멀티 디바이스 computation strategy
 - Keras API와 custom training loop 모두 지원.
 - 코드 자체에 적은 변화.

Distribution Strategy

tf.distribute.Strategy 타입들


- One Device Strategy
- Mirrored Strategy
- Parameter Server Strategy
- Multi-Worker Mirrored Strategy
- Central Storage Strategy
- TPU Strategy


Join TensorFlow at Google I/O, May 11-12 [Register now](#)

TensorFlow > API > TensorFlow Core v2.8.0 > Python

Was t

tf.distribute.Strategy

 TensorFlow 1 version

 View source on GitHub

A state & compute distribution policy on a list of devices.

```
tf.distribute.Strategy(  
    extended  
)
```

See [the guide](#) for overview and examples. See [tf.distribute.StrategyExtended](#) [tf.distribute](#) for a glossary of concepts mentioned on this page such as "per-reduce".

In short:

To use it with Keras [compile](#) / [fit](#), [please read](#)

One Device Strategy



Single device: 분산처리 x

일반적으로 distributed training 셋업을 하고 분산 strategy 실행 전에 테스트용으로 사용.

```
strategy = tf.distribute.OneDeviceStrategy(device="/gpu:0")

with strategy.scope():
    v = tf.Variable(1.0)
    print(v.device) # /job:localhost/replica:0/task:0/device:GPU:0

def step_fn(x):
    return x * 2

result = 0
for i in range(10):
    result += strategy.run(step_fn, args=(i,))
print(result) # 90
```

Mirrored Strategy



1개의 머신에서 여러개의 **GPU**를 사용할 수 있을 때 사용.

GPU당 replica를 생성해서 **variable**들을 미러링.

Weight 업데이트는 GPU간 커뮤니케이션 필요 (efficient) -> “AllReduce” 알고리즘.

```
tf.distribute.MirroredStrategy(  
    devices=None, cross_device_ops=None  
)
```

Parameter Server Strategy



각 노드 당 두가지 type:

- Worker 노드
- Parameter Server 노드

Parameter server는 shared variable들을 저장하며, Worker 노드들이 값을 업데이트 함.

Asynchronous parallelism을 수행.

```
tf.distribute.experimental.ParameterServerStrategy(  
    cluster_resolver, variable_partitioner=None  
)
```

- 현재 (TF2.8) [Experimental API](#)
- [Colab 튜토리얼](#)

Keras Multi Worker Strategy



[Colab Link](#)

Break +

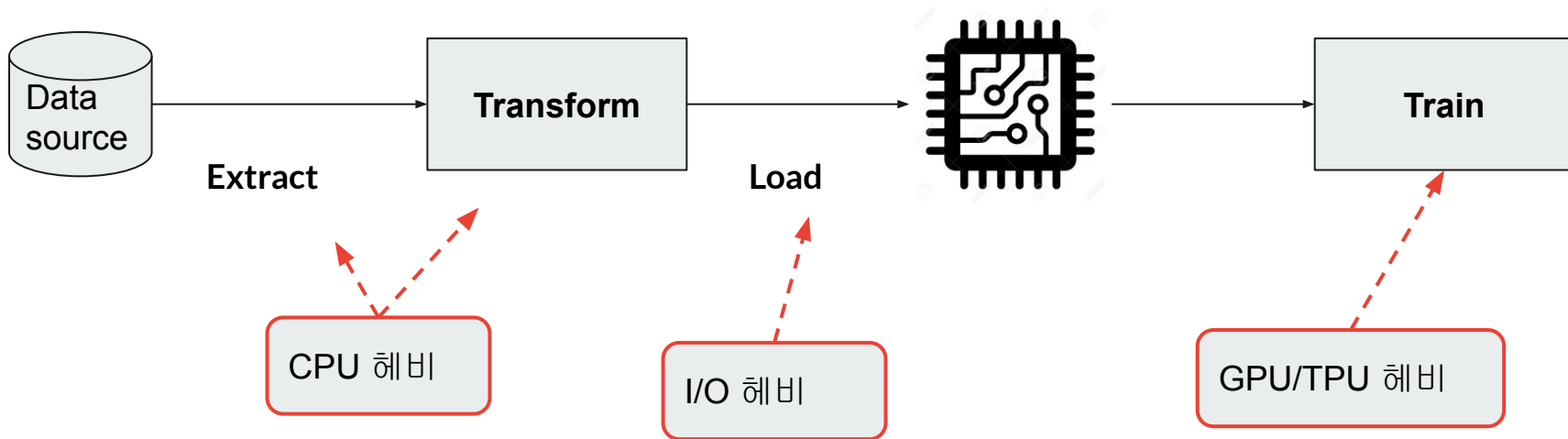
Ask Me Anything: 링크

ETL + ML 파이프라인 최적화

ETL with tf.data

ETL Pipeline + Model training

- Raw 데이터를 ML이 사용할 수 있는 안정적인 데이터로 처리하는 파이프라인.
- Extract (E), Transform (T), Load (L)
- 마지막 Loading된 데이터로 학습 시작.



ETL with tf.data

Extract

소스에서 데이터를 추출.

- Partial / Full extraction
- Partial extract: 업데이트 되었을 때 변경된 부분만 추출
- Full extract: 변경되었는지 파악 못할 경우 전부 추출

Local: HDD / SSD
Remote: HDFS, GCS, S3

Transform

데이터 정리, 매핑, 스키마로 변환.
(* Opt: 준비된 데이터를 staging data로 변환)

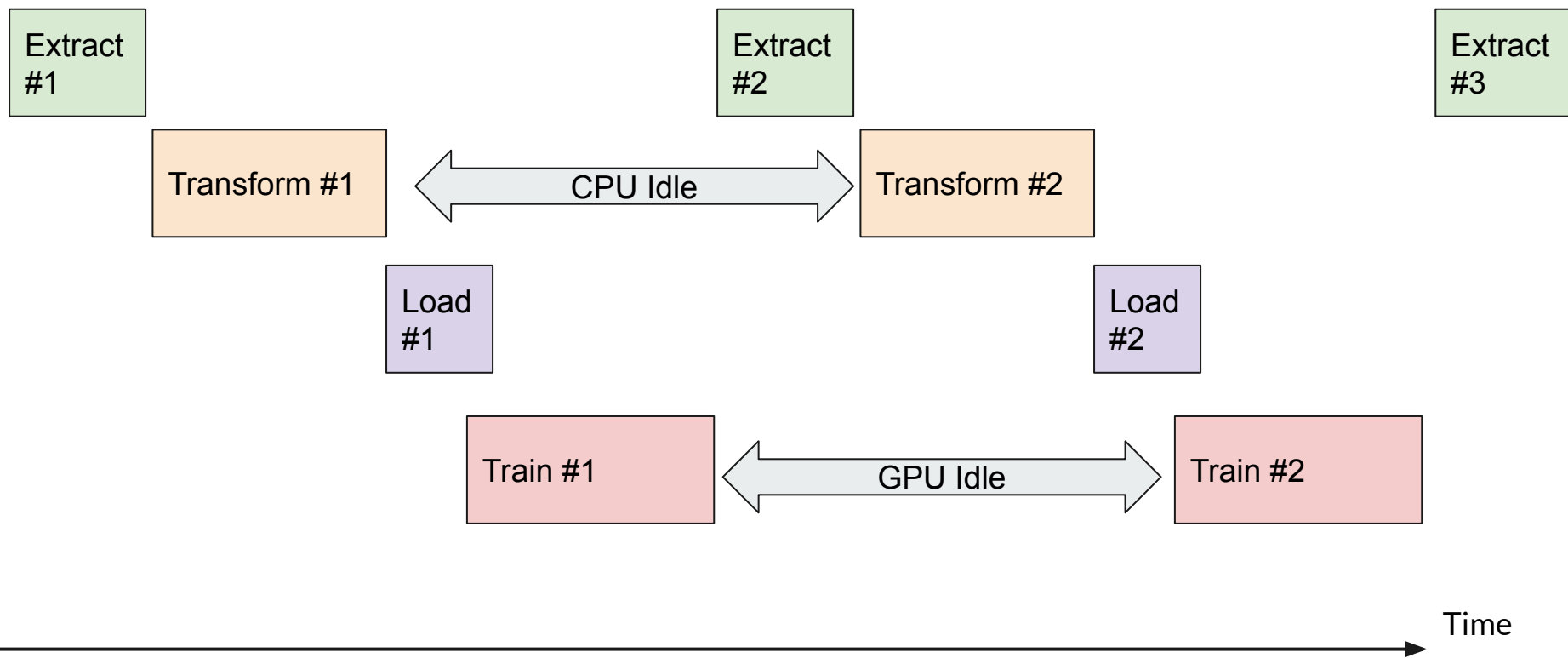
- Shuffling / Batching
- Augmentation, Vectorization

Load

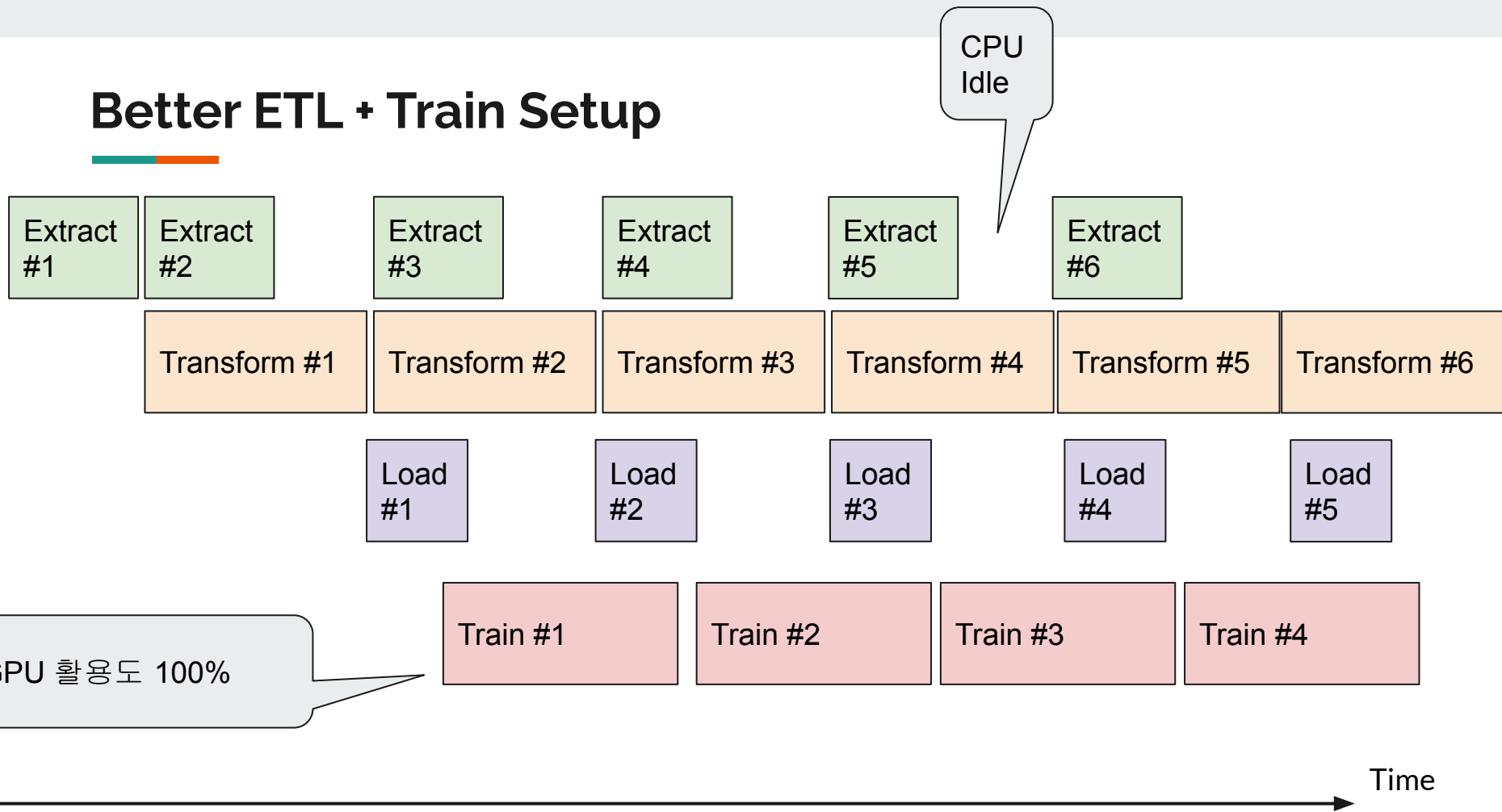
변환된 데이터를 staging에서부터 저장 / 로딩.

accelerator로 로딩

Poor ETL + Train Setup

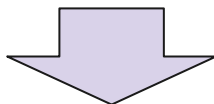


Better ETL + Train Setup



Pipeline을 통해 Compute Efficient

CPU	Prep 1	Idle	Prep 2	Idle	Prep 3	Idle
GPU/TPU	Idle	Train 1	Idle	Train 2	Idle	Train 3



CPU	Prep 1	Prep 2	Prep 3	Prep 4
GPU/TPU	Idle	Train 1	Train 2	Train 3

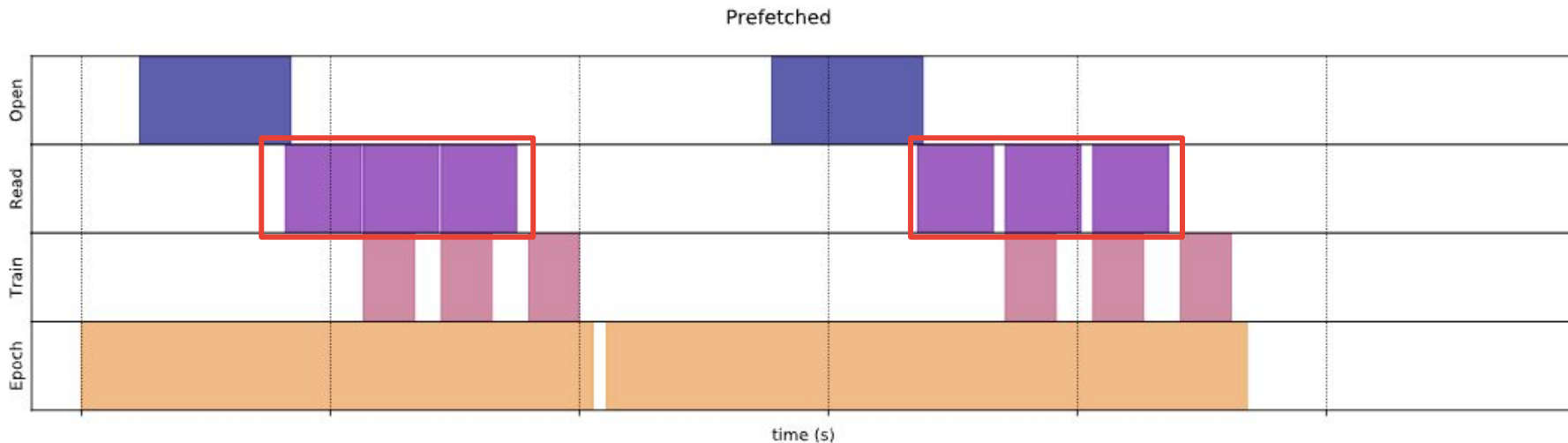
Maximize utility =
Minimize Idle Time

Pipeline을 통해 Compute Efficient



- Idle 한 리소스를 줄이도록 최적화
- 다양한 시스템 최적화 옵션:
 - Prefetching
 - Extract / Transform 병렬처리
 - Caching

Prefetching



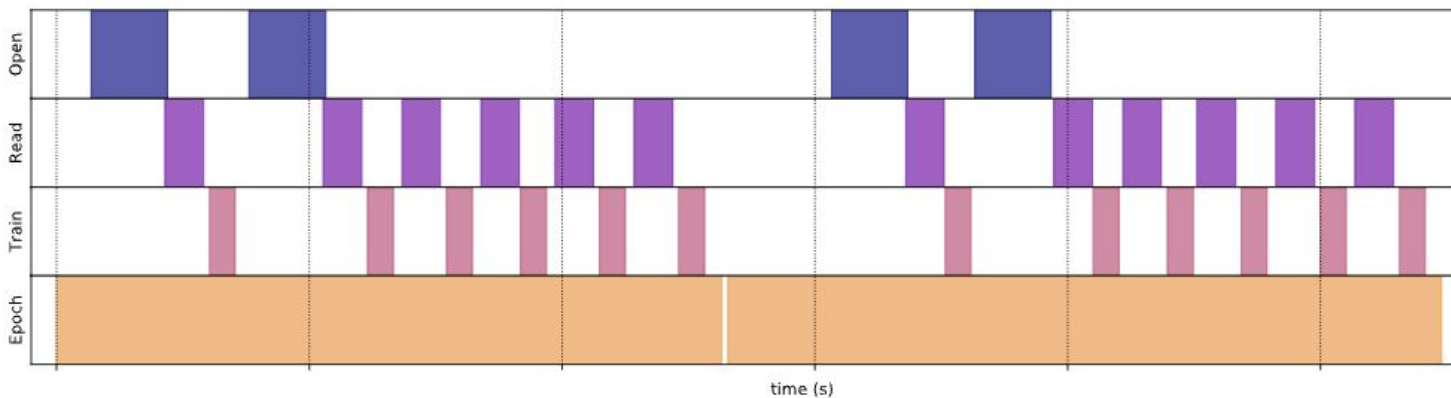
Training step 동안 read를 먼저 작업.

```
benchmark(  
    ArtificialDataset()  
    .prefetch(tf.data.AUTOTUNE)  
)
```

Extract 병렬처리

데이터 소스가 리모트에 존재할 경우 (GCS, HDFS), 로컬 데이터 (HDD, SSD)보다 현저히 액세스가 느려진다.

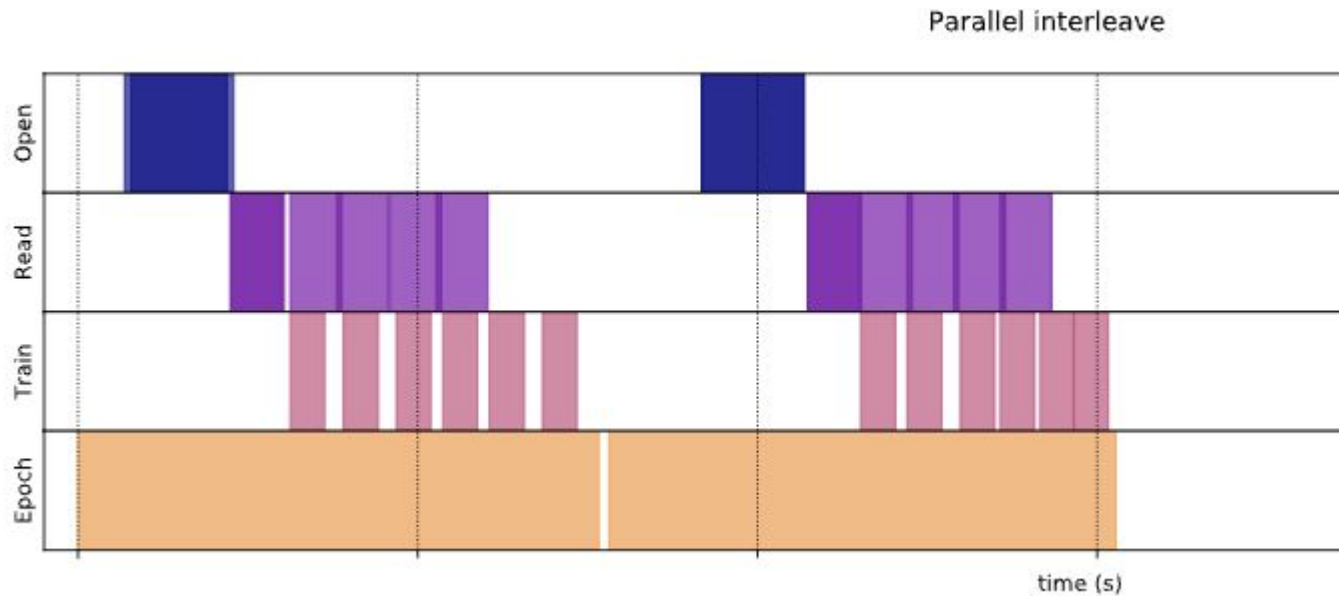
- **Time-to-first-byte:** RPC를 통해 데이터를 읽어올때 “첫번째 바이트”를 읽기까지의 시간이 오래걸린다 (network ping speed)
- **Read throughput:** Bandwidth / Throughput이 높은 네트워크라도, 한개의 파일만 읽을 경우 그 bandwidth를 다 쓰지 못한다.



Idle time 소모가
많음!

Extract 병렬처리

→ 여러개의 데이터소스를 쓸 때, **Interleave**를 사용.

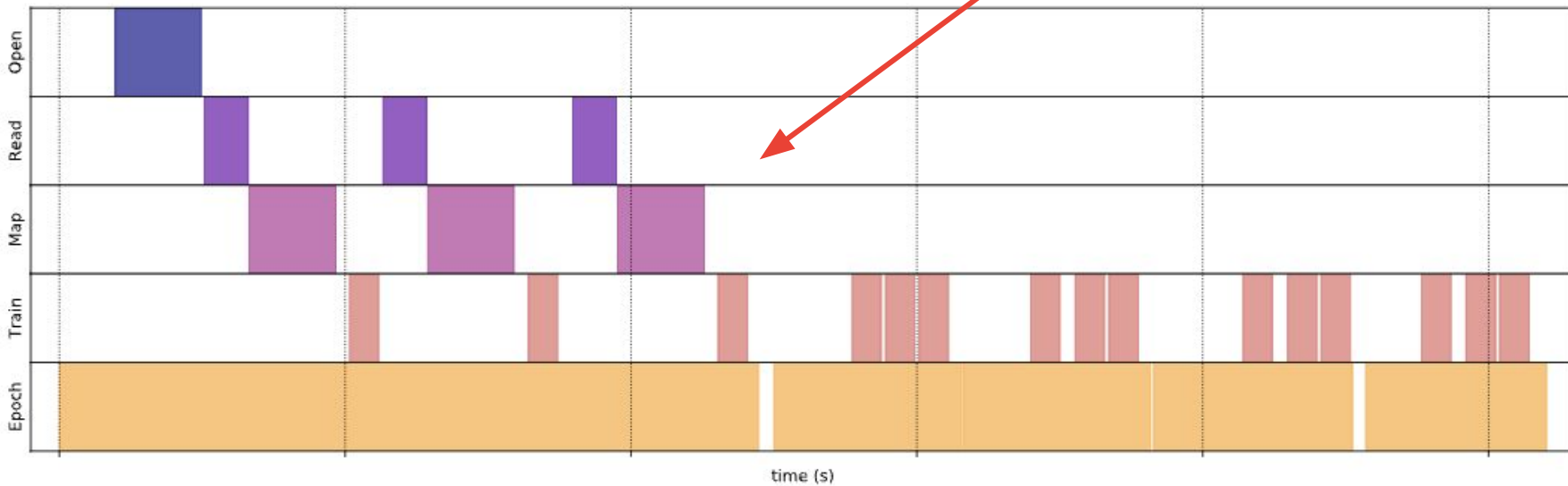


Caching

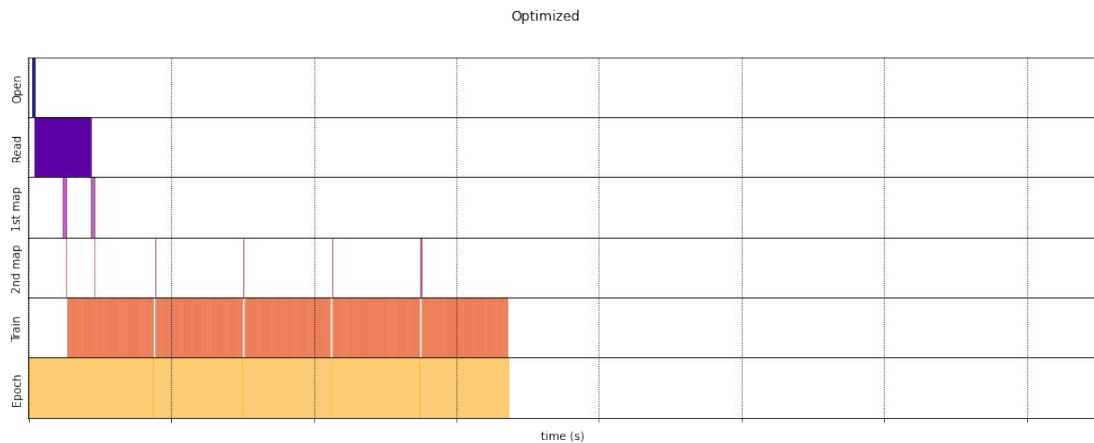
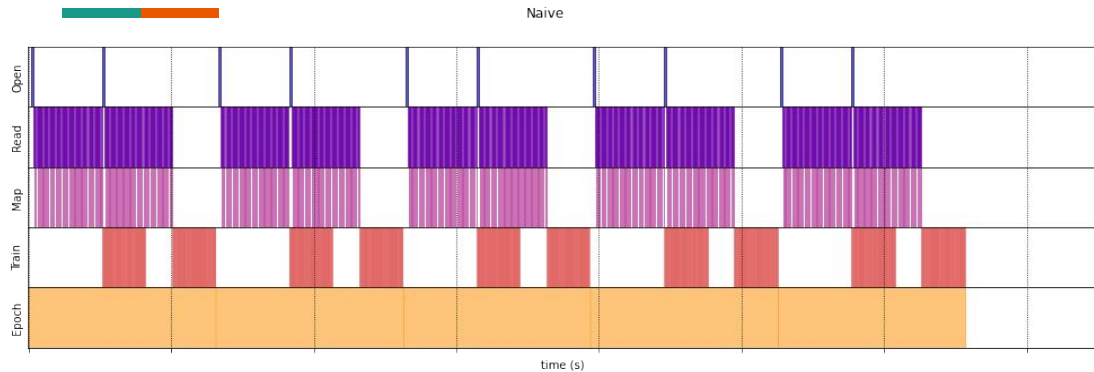
Transform시 데이터를 caching하여 로컬스토리지에 저장한다.

Read/Map이 첫번째 epoch에만 실행.

Cached dataset



Pipeline을 통해 Compute Efficient



- [Colab Example](#)
(Try 해보세요!)

Ask Me Anything